MONGO DB

Ferramentas:

- Docker(mongo)

O que é?

▼ O MongoDB é um sistema de gerenciamento de banco de dados NoSQL (not only SQL) que difere significativamente dos sistemas de gerenciamento de banco de dados SQL tradicionais. Abaixo estão alguns tópicos que podem ajudar a entender melhor as diferenças entre MongoDB e SQL.

1.Estrutura de Dados

- SQL: estrutura de dados rígida e altamente organizada com tabelas, linhas e colunas
- MongoDB: estrutura de dados flexível com documentos e campos que podem variar de tamanho e conteúdo

2. Linguagem de consulta

- SQL: linguagem estruturada para consulta e manipulação de dados relacionais
- MongoDB: utiliza uma linguagem de consulta baseada em JavaScript para consultar documentos em formato BSON (Binary JSON)

3. Escalabilidade

- SQL: escalabilidade vertical, limitada ao hardware disponível
- MongoDB: escalabilidade horizontal, distribuindo os dados em vários servidores para aumentar a capacidade de armazenamento

4. Integridade de dados

- SQL: garante a integridade dos dados através de restrições de chave primária, chave estrangeira e outras restrições de integridade referencial

- MongoDB: não possui restrições de integridade referencial, mas permite a validação de dados através de recursos de validação no nível do documento

5. Armazenamento de dados

- SQL: armazena dados em tabelas relacionais, geralmente utilizando o modelo ACID (Atomicidade, Consistência, Isolamento, Durabilidade)
- MongoDB: armazena dados em documentos BSON, utilizando o modelo BASE (Basically Available, Soft-state, Eventually Consistent)

6. Flexibilidade de esquema

- SQL: exige que o esquema seja definido antes da criação da tabela
- MongoDB: permite que o esquema seja alterado dinamicamente, sem afetar as outras partes do documento

7. Indexação de dados

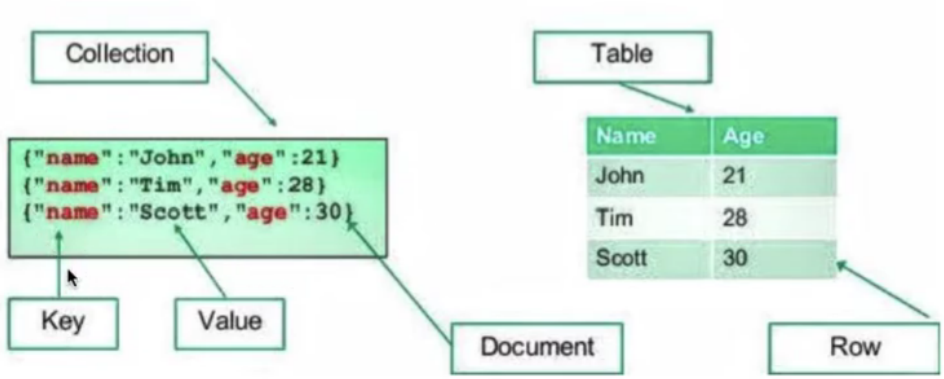
- SQL: utiliza índices para acelerar consultas em tabelas relacionais
- MongoDB: utiliza índices para acelerar consultas em coleções, permitindo uma consulta mais rápida em documentos específicos

Como está organizado?

1. **Coleções:**

    - No MongoDB, os documentos são agrupados em coleções. Uma coleção é análoga a uma tabela em um banco de dados relacional, mas com algumas diferenças significativas. Em vez de ter esquemas rígidos e tabelas com colunas predefinidas, as coleções no MongoDB são flexíveis e podem acomodar documentos com estruturas diferentes.

- No MongoDB, o tamanho máximo padrão para um documento BSON é de 16 megabytes (MB). Isso significa que cada documento armazenado em uma coleção do MongoDB não pode exceder 16 MB.



▼ Mongodb e arquivo Docker, configurando o ambiente.

```yaml
# Use root/example as user/password credentials
version: '3.1'

services:

  mongo:
    image: mongo
    container_name : mongoTeste
    restart: always
    ports:
      - 27017:27017
    environment:
      MONGO_INITDB_ROOT_USERNAME: root
      MONGO_INITDB_ROOT_PASSWORD: example

  mongo-express:
    image: mongo-express
    restart: always
    ports:
      - 8081:8081
    environment:
      ME_CONFIG_MONGODB_ADMINUSERNAME: root
      ME_CONFIG_MONGODB_ADMINPASSWORD: example
      ME_CONFIG_MONGODB_URL: mongodb://root:example@mongo:270
```

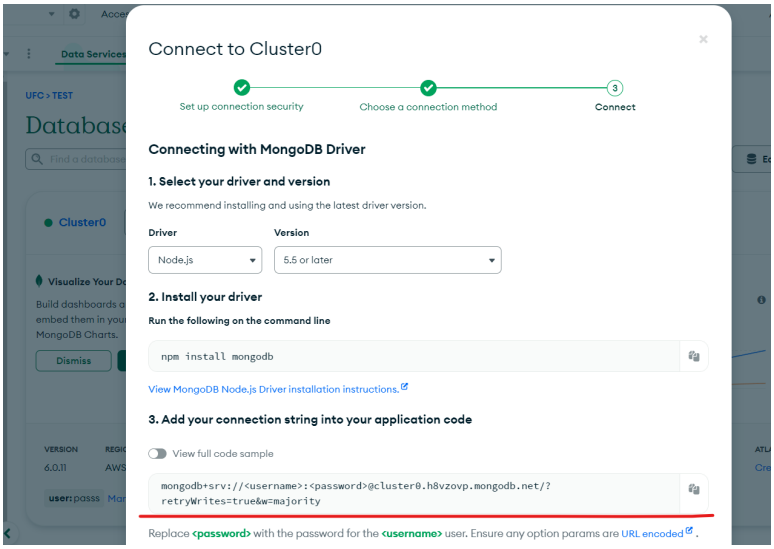
Passo a passo para iniciar o Mongo:

```
Abra o CMD e digite:
docker-compose up//Sobe o arquivo para o docker e ele cria o

docker ps //mostrar todos os containers

docker exec -it mongoTeste bash // executa o container mongoT

mongosh //Faz a conexão
```

```
use admin //entra como admin pra ter acesso ao mongo
db.auth('root', passwordPrompt()) // coloca a senha(example)
```

▼ Mongodb Atlas e Colab, configurando o ambiente (Nuvem)

Primeiramente, é necessário criar uma conta no Colab, pois é uma máquina virtual que executa Python. Em seguida, faça uma conta no MongoDB Atlas, um serviço gratuito do MongoDB que oferece um servidor gratuito. Após a criação da conta no MongoDB Atlas, obtenha a string de conexão necessária, a qual pode ser encontrada no local indicado na imagem:



Antes de começar a utilizar o Colab, é importante criar um Database Access no MongoDB Atlas, onde você definirá o usuário e a senha para autenticação. Em seguida, abra o Colab e realize a conexão conforme o código:

```
!pip install pymongo
from pymongo import MongoClient

# Substitua <"SEU USER"> e <"SUA SENHA">
```

```
connection_string = "mongodb+srv://"SEU USER":"SUA SENHA"@clu
client = MongoClient(connection_string)
```

COMANDOS BÁSICOS:

```
//COMANDOS BÁSICOS INICIAIS

db.version //Mostra as informações do DB
show dbs //Mostra todos os bancos
use "name_database" //para entrar em um certo db
db.getCollectionNames() //Mostra as collections
db."name_collection".find() //Mostra todos os arquivos
db."name_collection".insertOne({'**Exemple:** name: "Larry", age

//COMANDOS BÁSICOS COM PYMONGO

db = client['<YOUR_DB_NAME>'] //Acessar o banco de dados desejad
//Comandos
print(db.version())//Mostra as informações do DB
print(client.list_database_names())//Mostra todos os bancos
print(db.list_collection_names())//Mostra as collections
cursor = db."name_collection".find()//Mostra todos os arquivos n
for document in cursor:
    print(document)
db."name_collection".insert_one({'name': 'Larry', 'age': 123, 'g
```

Depois de termos o Mongo já pronto vamos para os comandos principais:

CRUD

Create → Criar um recurso

1. insertOne({})

```
// Exemplo de insertOne para inserir um único documento em uma c
db.clientes.insertOne({ nome: 'Maria', idade: 25 });
```

2. insertMany([{},{},{}])

> Adicionar: {ordered:false}) para quando um arquivo não dar certo os outros serem adicionados

```
db.produtos.insertMany([
  { nome: 'Produto 1', preco: 19.99 },
  { nome: 'Produto 2', preco: 29.99 },
  { nome: 'Produto 3', preco: 39.99 },
  { nome: 'Produto 4' }  // Este documento está faltando o campo
], { ordered: false });
```

Read → Ler o recurso

1. find()

```
//    O método find() recupera todos os documentos de uma coleçã
db.suaColecao.find();  // Exemplo básico de find() sem critério

//    É útil para visualizar os resultados de maneira mais organ
db.suaColecao.find().pretty();  // Exemplo de find() seguido de
```

2. findOne

```
// Encontrar um documento na coleção 'alunos' onde todos os docu
// e projetar apenas os campos 'nome' e 'materias' (excluindo '_
db.alunos.findOne(
  {/* TODOS */},          // Critério de consulta (nenhum critér
  { nome: 1, materias: 1, _id: 0 }  // Projeção: Incluir 'nome'
```

```
);

// Resultado do findOne:
{
   nome: 'Antonio',
   materias: { portugues: 8.9, matematica: 10, historia: 4 }
}
```

Update→ Atualizar o recurso

1. updateOne
2. updateMany

```
// Atualiza o primeiro documento onde o campo "name" é igual a '
db.usuarios.updateOne(
  { name: "John" },        // Critério de consulta
  { $set: { status: "ativo" } }   // Operação de atualização uti
);

// Atualiza vários documentos na coleção "crud" onde o campo "a'
// Utiliza o operador $set para definir o valor do campo "h" cor
db.crud.updateMany({a: 123}, {$set: {h: 20}});
//RESULTADO
{
  acknowledged: true,   // Indica se a operação foi reconhecida
  insertedId: null,     // O ID do documento inserido. Neste cas
  matchedCount: 3,      // Número de documentos que correspondem
  modifiedCount: 3,     // Número de documentos que foram efetiv
  upsertedCount: 0      // Número de documentos inseridos pela c
}
```

3. replaceOne

```
//COMANDO
db.crud.replaceOne({a:123},{outro:"documento"})
```

```
//PRINT
{ _id: ObjectId("650a5eba60e33f961bfff687"), a: 555, d: 555 },
{ _id: ObjectId("650a5ebe60e33f961bfff688"), outro: 'documento'
{ _id: ObjectId("650a5ed660e33f961bfff689"), a: 123, h: 20 },
```

Delete → Deletar o recurso

1. deleteOne
2. deleteMany
3. db.collection.drop()

```
db.alunos.deleteOne({ name: "Larry" }); //Remove um documento c
db.pessoas.deleteMany({ idade: { $gt: 25 } }); //Remove todos c
db."nameCollection".deleteMany({}) //Apaga todos os dados da co
db."nameCollection".drop() //Deleta a Collection e todos os seu
```

Relacionamentos

Em sistemas de gerenciamento de banco de dados (SGBDs) convencionais, é comum utilizar chaves estrangeiras para estabelecer relacionamentos entre tabelas, e os dados são organizados de acordo com o tipo e a lógica adotada. No entanto, no MongoDB, frequentemente optamos por consolidar os dados, evitando buscas complexas, como JOINs em vários arquivos. Esse tipo de operação pode ser demorado e exigir recursos significativos do servidor.

No MongoDB, a abordagem difere, pois a plataforma visa aprimorar a eficiência ao agregar os dados sempre que possível. A ideia é evitar a necessidade de JOINs, uma vez que essa operação é conhecida por seu custo considerável. Ao consolidar os dados, o MongoDB busca simplificar as consultas e otimizar o desempenho, priorizando uma estrutura que favoreça a recuperação direta das informações necessárias, sem depender excessivamente de operações de junção, que podem ser dispendiosas em termos de tempo e recursos do servidor.

Modelo ER

No MongoDB, assim como em outros sistemas de gerenciamento de banco de dados, podemos modelar diferentes tipos de relacionamentos entre documentos. Os três tipos principais de relacionamentos são:

1. **1:1 (Um para Um):**
    - Este relacionamento envolve uma correspondência única entre dois documentos. Cada documento em uma coleção está vinculado a exatamente um documento em outra coleção e vice-versa. Isso é comumente implementado incorporando os dados diretamente em um documento ou referenciando um documento pelo "_id" no outro.
2. **1:n (Um para Muitos):**
    - Este relacionamento representa uma correspondência onde um documento em uma coleção pode estar relacionado a vários documentos em outra coleção. Isso é frequentemente alcançado usando a abordagem de referência, onde um documento contém uma lista de _ids relacionados a outros documentos em uma coleção.
3. **n:n (Muitos para Muitos):**
    - Neste relacionamento, múltiplos documentos em uma coleção podem estar relacionados a múltiplos documentos em outra coleção. Para implementar isso no MongoDB, é comum usar a abordagem de referência cruzada, onde os documentos contêm listas de _ids referentes a documentos em outras coleções.

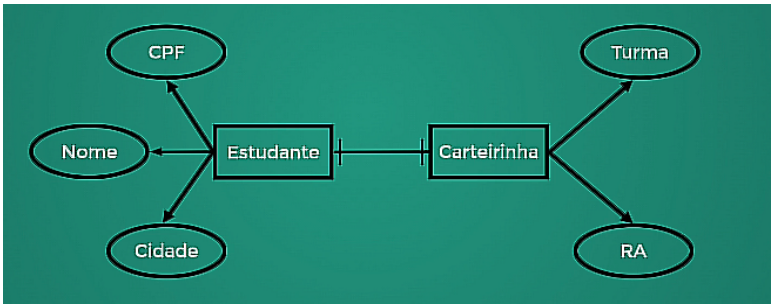
- **Embedded Document**

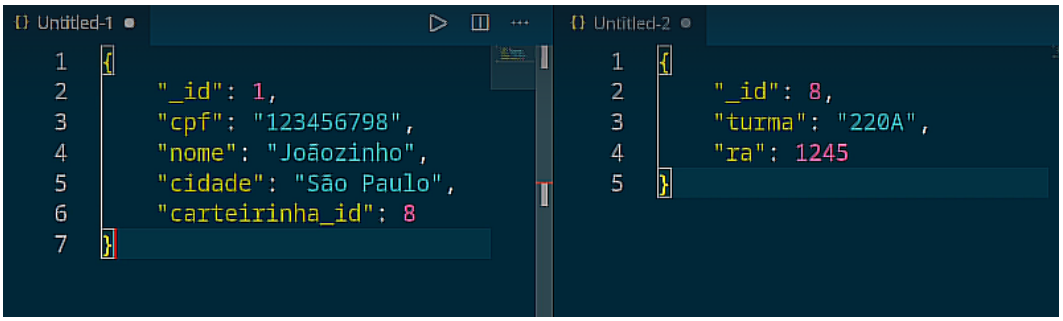
Em MongoDB, um "Embedded Document" (Documento Incorporado) refere-se a um documento BSON que é incorporado dentro de outro documento BSON. Em vez de armazenar referências a documentos em coleções separadas (como em um relacionamento de referência), os documentos incorporados são incluídos diretamente no documento pai.

Um para um (1:1)

- Ao empregar uma Referência ou SK (Secondary Key) em um banco de dados NoSQL, a necessidade de realizar um JOIN se apresenta, indicando um possível uso inadequado do paradigma NoSQL.

Exemplo de banco relacional: Estudante e carteirinha relacionamento.





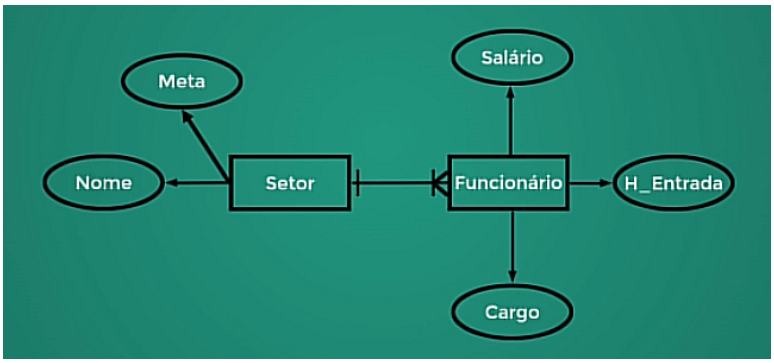
No não relacional(MongoDB) ficaria assim:

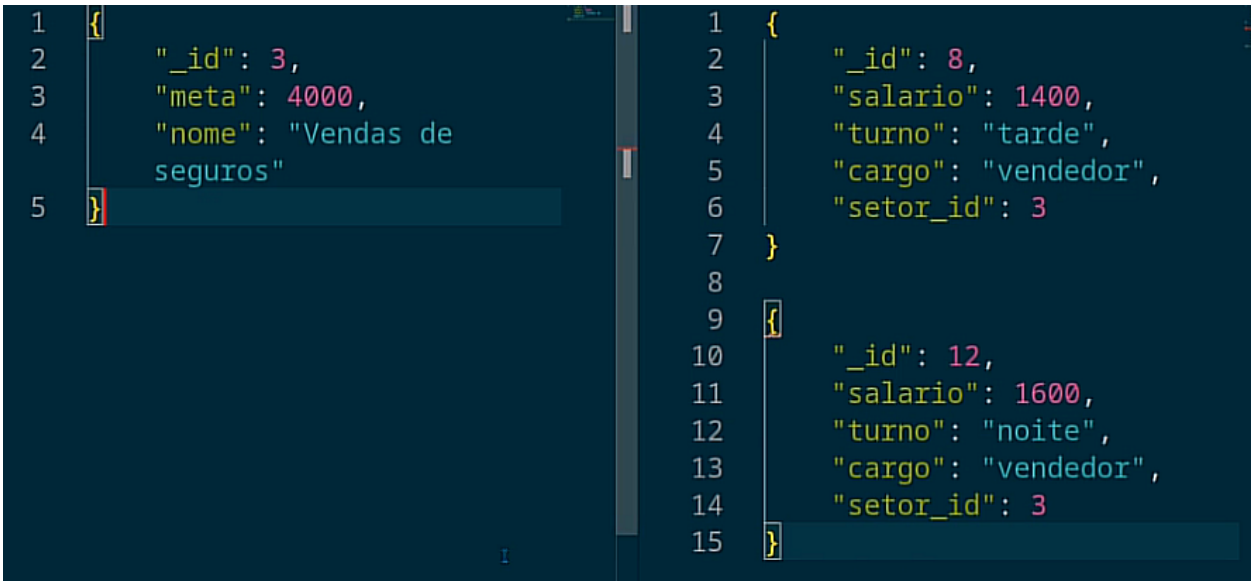
```
Embedded documents (recomendado)
{
    "_id": 1,
    "cpf": "123456798",
    "nome": "Joãozinho",
    "cidade": "São Paulo",
    "carteirinha": {
        "_id": 8,
        "turma": "220A",
        "ra": 1245
```

```
    }
}
//Nós juntamos os documentos
```

Um para muitos(1:n)

Empresa com Setor e Funcionário





```
{
    "_id": 3,
    "meta": 4000,
    "nome": "Vendas de
    seguros"
}
```

```
{
    "_id": 8,
    "salario": 1400,
    "turno": "tarde",
    "cargo": "vendedor",
    "setor_id": 3
}

{
    "_id": 12,
    "salario": 1600,
    "turno": "noite",
    "cargo": "vendedor",
    "setor_id": 3
}
```

Vai depender de quantos Funcionários tem na empresa, pois se for muitos mesmo pode passar dos 16mb que o mongo suporta.

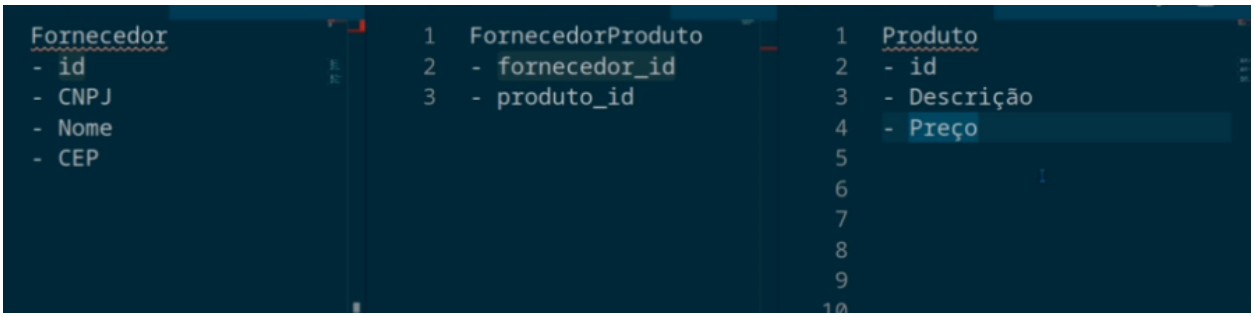
Passando para NOSQL(a grade sacada é não usar o Join)

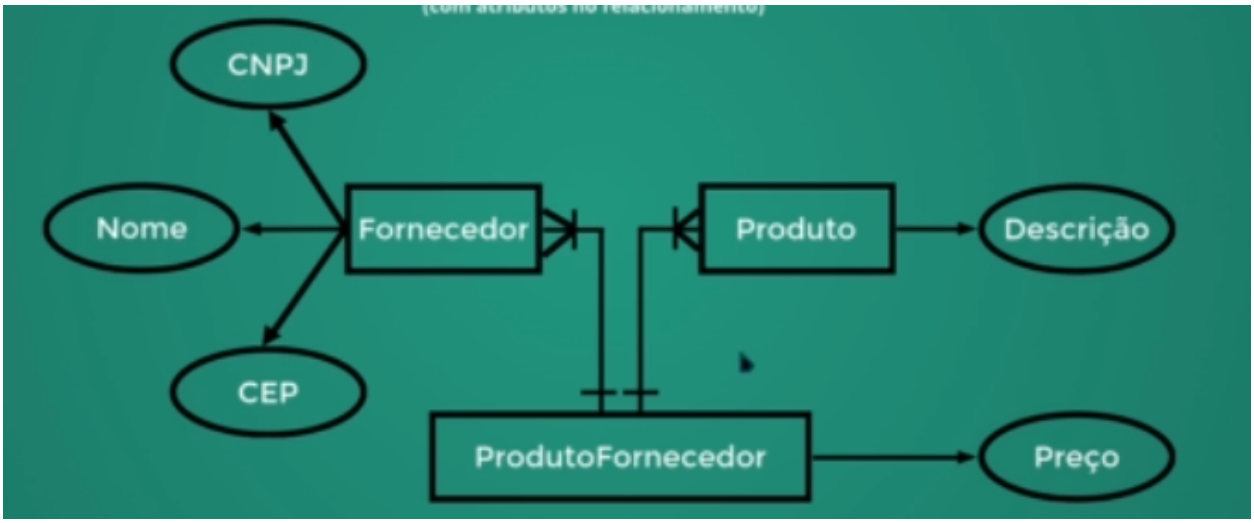
```
{
    "_id": 3,
```

```
    "meta": 4000,
    "nome": "Vendas",
    "funcionarios": [
        {
            "_id": 8,
            "salario": 1400,
            "turno": "tarde",
            "cargo": "vendedor"
        },
        {
            "_id": 12,
            "salario": 1600,
            "turno": "noite",
            "cargo": "vendedor"
        }
    ]
}
```

Muitos para muitos(n:n)

No SQL cria-se outra tabela





No NoSQl nós não criamos outra tabela, colocamos as referências:

```
{
    "_id": "f04",
    "cnpj": "165486984",
    "nome": "Fornecedor Legal",
    "cep": "1098465",
    "produto_ids": ["p16", "p21"]
}

{
    "_id": "f07",
    "cnpj": "98498151",
    "nome": "Fornecedor Maneiro",
    "cep": "198498",
    "produto_ids": ["p21", "p47"]
}
```

```
{
    "_id": "p16",
    "descricao": "Panela",
    "preco": 45.50,
    "fornecedor_ids": ["f04"]
}

{
    "_id": "p21",
    "descricao": "Prato",
    "preco": 14,
    "fornecedor_ids": ["f04", "f07"]
}

{
    "_id": "p47",
    "descricao": "Faqueiro",
    "preco": 127.46,
    "fornecedor_ids": ["f07"]
}
```

Se tivermos um atributo na relação não podemos fazer assim. O melhor a fazer é trabalhar com a redundância de dados, não termos mais “fornecedor_ids” nem “produto_ids” no lugar colocamos um array de documentos, com os dados que necessitamos. Cuidado!! Ao alterar o valor de preço no exemplo tem que alterar nos dois lados.

```
// Collection Fornecedores
{
    "_id": "f04",
    "cnpj": "165486984",
    "nome": "Fornecedor Legal",
    "cep": "1098465",
    "produtos": [
        {
            "_id": "p16",
            "preco": 46.50
        },
        {
            "_id": "p21",
            "preco": 12
        }
    ]
}

{
    "_id": "f07",
    "cnpj": "98498151",
    "nome": "Fornecedor Maneiro",
    "cep": "198498",
    "produtos": [
        {
            "_id": "p21",
            "preco": 16
        },
        {
            "_id": "p47",
            "preco": 127.46
        }
    ]
}
```

```
// Collection Produtos
{
    "_id": "p16",
    "descricao": "Panela",
    "fornecedores": [
        {
            "_id": "f04",
            "preco": 46.50
        }
    ]
}

{
    "_id": "p21",
    "descricao": "Prato",
    "fornecedores": [
        {
            "_id": "f04",
            "preco": 12
        },
        {
            "_id": "f07",
            "preco": 16
        }
    ]
}

{
    "_id": "p47",
    "descricao": "Faqueiro",
    "fornecedores": [
        {
            "_id": "f07",
            "preco": 127.46
        }
    ]
}
```

Duplicar ou não duplicar dados?

É crucial considerar a aplicação em questão. Cada situação é única, mas ao duplicar dados, é preferível que o banco de dados seja mais utilizado para leitura do que para atualização dos valores. É importante lembrar que, ao duplicarmos os dados, devemos garantir que qualquer atualização seja refletida em todos os lugares onde esses dados estão presentes, para evitar erros na aplicação.

Se o seu banco de dados é atualizado frequentemente e você está lidando com um grande volume de atualizações, pode ser benéfico considerar abordagens tradicionais, semelhantes às usadas em bancos de dados relacionais, dependendo dos requisitos específicos da sua aplicação. Vou destacar algumas considerações:

Consultas (queries) em MongoDB

As consultas, ou queries, no MongoDB abrangem desde as mais básicas, envolvendo apenas uma condição e ordenação de resultados, até operações avançadas de agregação que podem ser comparadas aos JOINS em bancos de dados relacionais.

Exemplo de consultas:

```
//Selecionar todos os documentos de uma coleção:
db.clientes.find({});
//Filtrar documentos com uma condição:
db.produtos.find({ categoria: 'eletrônicos' });
//Ordenar resultados:
db.pedidos.find().sort({ data_pedido: -1 });
//Realizar uma agregação (equivalente a JOIN):
db.clientes.aggregate([
  {
    $lookup: {
      from: "pedidos",
      localField: "_id",
      foreignField: "id_cliente",
      as: "pedidos"
```

```
    }
  },
  {
    $unwind: "$pedidos"
  },
  {
    $project: {
      nome: 1,
      numero_pedido: "$pedidos.numero_pedido",
      valor: "$pedidos.valor"
    }
  }
]);
```

Neste exemplo, `$lookup` realiza uma operação de junção semelhante ao JOIN em SQL, e `$project` projeta os campos desejados no resultado.

Para melhor entendimento em SQL ficaria assim:

```
SELECT clientes.nome, pedidos.numero_pedido, pedidos.valor
FROM clientes
INNER JOIN pedidos ON clientes.id = pedidos.id_cliente;
```

As consultas no MongoDB não se limitam a isso; elas continuam a oferecer uma variedade de operadores. Abaixo, apresento mais alguns exemplos. Para visualizar como esses operadores são utilizados, basta acessar a documentação oficial do MongoDB.

▼ Operadores de Comparação:

`$eq` - Igual

`$ne` - Diferente de

`$gt` - Maior que

`$gte` - Maior ou igual a

`$lt` - Menor que

`$lte` - Menor ou igual a

`$in` - Igual a qualquer valor em um conjunto especificado

`$nin` - Diferente de todos os valores em um conjunto especificado

Operadores Lógicos:

`$and` - Operador lógico AND

`$or` - Operador lógico OR

`$not` - Negar uma expressão lógica

`$nor` - Operador lógico NOR (Negação de OR)

Operadores Elementares:

`$exists` - Verifica se um campo existe

`$type` - Seleciona documentos com um tipo de dado específico

Operadores de Expressão Regular:

`$regex` - Realiza correspondência de expressão regular em strings

Operadores de Array:

`$all` - Corresponde a arrays que contêm todos os elementos especificados

`$elemMatch` - Corresponde a documentos que contenham pelo menos um elemento que satisfaça todas as condições especificadas

Operadores de Projeção:

`$project` - Especifica os campos a serem incluídos ou excluídos em uma consulta

Operadores de Agregação:

`$group` - Agrupa documentos para realizar operações de agregação

`$match` - Filtra os documentos para permitir apenas aqueles que atendem a determinadas condições

`$sort` - Classifica os documentos com base nos campos especificados

`$limit` - Limita o número de documentos retornados em uma consulta

`$skip` - Pula uma quantidade específica de documentos em uma consulta
Operadores de Atualização:

`$set` - Atualiza o valor de um campo existente ou adiciona um novo campo

`$unset` - Remove um campo de um documento

`$inc` - Incrementa o valor de um campo numérico

`$push` - Adiciona um valor a um array

`$addToSet` - Adiciona um valor a um array se ele não estiver presente

`$pull` - Remove todos os elementos de um array que correspondem a uma condição especificada

Introdução a Schemas

O conceito de "schema" desempenha um papel fundamental em sistemas de gerenciamento de banco de dados, proporcionando uma estrutura organizada e predefinida para os dados armazenados. Um schema define a forma como os dados são organizados, quais tipos de dados podem ser armazenados e quais relacionamentos existem entre eles.

A validação deve ocorrer no backend, mas é necessário e útil também termos validações no banco de dados. O MongoDB não é tão voltado para essa área, mas ele oferece as seguintes ferramentas:

```
db.createCollection() //Por que usar esse comando já que o mongo
db.createCollection(name, options) // name = Nome da collection

//Syntax, Retirado da documentação do mongo
db.createCollection( <name>,
    {
      capped: <boolean>,
      timeseries: {                  // Added in MongoDB 5.0
         timeField: <string>,        // required for time series
         metaField: <string>,
         granularity: <string>,
         bucketMaxSpanSeconds: <number>,  // Added in MongoDB 6
         bucketRoundingSeconds: <number>  // Added in MongoDB 6
      },
      expireAfterSeconds: <number>,
      clusteredIndex: <document>,  // Added in MongoDB 5.3
      changeStreamPreAndPostImages: <document>,  // Added in Mon
      size: <number>,
      max: <number>,
      storageEngine: <document>,
      validator: <document>, //VAMOS FOCAR NESSE!
      validationLevel: <string>,
      validationAction: <string>,
      indexOptionDefaults: <document>,
      viewOn: <string>,
      pipeline: <pipeline>,
      collation: <document>,
      writeConcern: <document>
    }
  )
```

Como fazemos uma collection com validation?

Documentação:https://www.mongodb.com/docs/manual/core/schema-validation/specify-json-schema/#std-label-schema-validation-json

```
db.createCollection("carros", { //Vamos passar o objeto validat
   validator: { //documento dentro do documento
      $jsonSchema: { //Schema do nosso Json, é outro objeto
         bsonType: "object", //TIPO
         title: "Student Object Validation", //Descrição do Sch
         required: [  "model", "year"], //PARA GARANTIR QUE A C
         properties: {
            model: {
               bsonType: "string",
               description: "O Nome é necessário e deve ser uma
               minLength: 3 //A String name não pode ter menos
            },
            year: {
               bsonType: "int", //cuidado tem que especificar,
               minimum: 2017, //Não pode ser usado com string
               maximum: 3017,
               description: "'year' é necessário e deve estar e
            },
            peso: {
               bsonType: [ "number" ],
                              minLength: 3,
               description: "'nota' tem que ser double se 'nota
}  }  }  } } )
```

Nesse caso o insert ficaria assim:

```
db.carros.insert({    "model" : "fusca",
                                          "year": NumberInt(20
                                          "medeBy":"Fiat",
                                          "peso": 342.2
})
```

Criação da tabela de usuários

```
db.createCollection("usuarios", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      title: "Objeto de validação para usuários",
      required: ["senha", "cpf", "email", "cargo", "nome", "role
      properties: {
        senha: {
          bsonType: "string",
          description: "Senha obrigatória!",
          minLength: 6,
          maxLength: 64
        },
        cpf: {
          bsonType: "string",
          minLength: 11,
          maxLength: 11,
          description: "CPF obrigatório!"
        },
        email: {
         bsonType: "string",
         description: "Email inválido!",
         pattern: "^[\w-\.]+@([\w-]+\.)+[\w-]{2,4}$"
        },
        cargo: {
         bsonType: "string",
         description: "Cargo é obrigatório!"
        },
        nome: {
         bsonType: "string",
         description: "Nome é obrigatório!"
        },
        role: {
         bsonType: "int",
```

```
          description: "Role é obrigatório!"
        },
        municipio_id: {
          bsonType: "int",
          description: "Código do município é obrigatório!"
        }
      },
    },
  },
})
```

Criação da tabela de informações sobre os descritores do SPAECE

```
db.createCollection("info_descritores", {
  validator: {
    $jsonSchema: {
      bsonType: "object",
      title: "Objeto de validação para informações dos descritor
      required: [
                "cod_descritor",
                "topico",
                "descricao",
                "cod_disciplina",
                "cod_etapa"
            ],
      properties: {
        cod_descritor: {
          bsonType: "string",
          description: "Código do descritor obrigatório!",
          minLength: 1,
          maxLength: 3
        },
        topico: {
          bsonType: "string",
          minLength: 1,
          maxLength: 255,
```

```
            description: "Tópico obrigatório!"
        },
        descricao: {
            bsonType: "string",
                description: "Descrição obrigatória!",
                    minLength: 1,
            maxLength: 255,
        },
        cod_disciplina: {
            bsonType: "number",
            description: "Código da disciplina obrigatório!",
                    minLength: 1,
                    maxLength: 1
        },
        cod_etapa: {
            bsonType: "number",
            description: "Código da etapa obrigatório!"
                    minLength: 1,
                    maxLength: 1
        }
      },
    },
  },
})
```